Scalable Hardware Support for Conditional Parallelization

Zheng Li
INRIA Saclay
Orsay, France
zheng.x.li@inria.fr

Olivier Certner
ST Microelectronics & INRIA Saclay
Orsay, France
olivier.certner@inria.fr

Jose Duato
Polytechnic University of Valencia
Valencia, Spain
jduato@disca.upv.es

Olivier Temam
INRIA Saclay
Orsay, France
olivier.temam@inria.fr

ABSTRACT

Parallel programming approaches based on task division/-spawning are getting increasingly popular because they provide for a simple and elegant abstraction of parallelization, while achieving good performance on workloads which are traditionally complex to parallelize due to the complex control flow and data structures involved. The ability to quickly distribute fine-granularity tasks among many cores is key to the efficiency and scalability of such division-based parallel programming approaches. For this reason, several hardware supports for work stealing environments have already been proposed. However, they all rely on a central hardware structure for distributing tasks among cores, which hampers the scalability and efficiency of these schemes.

In this paper, we focus on conditional division, a division-based parallel approach which provides the additional benefit, over work-stealing approaches, of releasing the user from dealing with task granularity and which does not clog hardware resources with an exceedingly large number of small tasks. For this type of division-based approaches, we show that it is possible to design hardware support for speeding up task division that entirely relies on *local* information, and which thus exhibits good scalability properties.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Design, Performance

Keywords

Multicore, conditional parallelization, hardware support

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PACT'10, September 11–15, 2010, Vienna, Austria.
Copyright 2010 ACM 978-1-4503-0178-7/10/09 ...$10.00.

1. INTRODUCTION

Because the advent of multi-cores may ultimately force a large fraction of programmers to use and write parallel codes, architecture research has been increasingly focused on facilitating parallel programming. To date, a lot of attention has been devoted to the issue of memory coherence, e.g., transactional memory [16]. In this paper, we focus on the issues of task granularity, load balancing and task mapping, which are largely orthogonal to memory coherence issues.

Programming models attempt to present an idealized and simplified view of the architecture to the programmer. Among the many different approaches, programming models based on tasks divisions are gaining traction. Cilk [8] introduced the notion of *spawning* where parallelization is described as dividing a task into two parts, a simple and elegant abstraction of parallelization. The different cores are then load balanced by allowing each core to steal tasks from another core through the Cilk run-time system. Intel TBBs (Thread Building Blocks) [21] are largely based upon the same principles, except they also propose a higher programming abstraction which can hide the underlying spawning and work stealing strategies. However both models require the user to control task granularity: depending on where the spawning statements are inserted, either too fine tasks may be created where the overhead of parallelism will cancel the benefit of parallelization, or too coarse tasks may be created which cannot take advantage of a large number of cores. Worse, the trade-off is largely architecture-dependent (number of cores, core performance, communication costs, ...), making the programs less portable. CAPSULE [20] proposed to solve this issue by *conditionally* dividing tasks: a task is split in two only if there is a core available to host it and the division is deemed profitable; tasks are no longer greedily spawned and queued, waiting for execution or to be stolen by other cores. Provided that checking for free cores is very fast (a few cycles), such probes can be inserted almost anywhere in the code, including within innermost loops. As a result, programmers need no longer worry about task granularity and can insert division statements anywhere they see a parallelization opportunity, including very fine granularity; the run-time system later decides if it is possible and profitable to take advantage of this potential parallelism. Moreover, the more frequent the probes, the faster the load is balanced among cores.

The performance of conditional division relies on the ability to quickly check if cores are available. This check can be

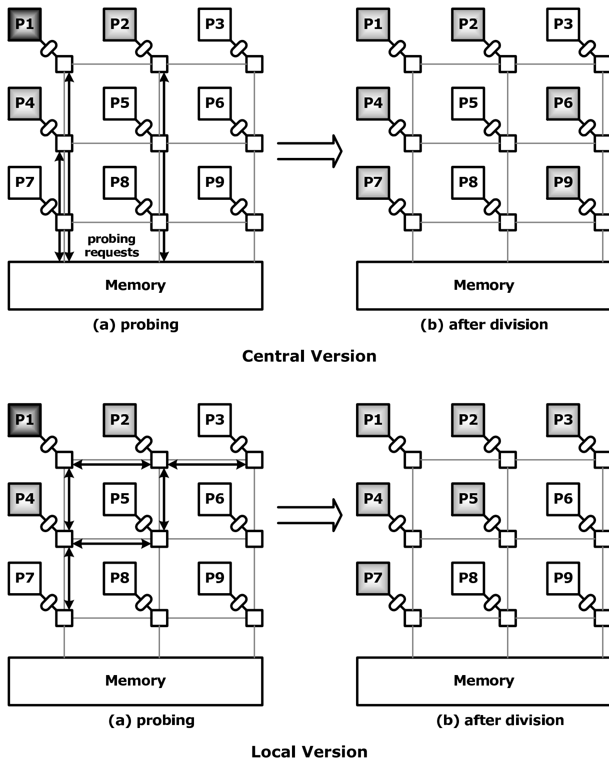

Figure 1: ***Central vs. Local throttling of tasks distribution; black arrows represent probing operations; grey shades represent pressure (darker means more pressure). In (a), all the probing requests from P1, P2 and P4 are sent to the central shared memory. In (c), P1 propagates the probing requests to its neighbors P2 and P4; P2 propagates the probing requests to P1, P3 and P5; P4 propagates the probing requests to P1, P3 and P5.***

either implemented through a shared variable indicating the number of available cores [10] or a hardware register [20]. However, in both cases, the information is *centralized* in a given memory location, as shown in Figure 1(a,b). And as the number of cores increases, the time required to access this centralized resource through the network increases, making checks exceedingly time-consuming, thereby voiding the benefits of conditional division. Reverting to work stealing is not a solution either: besides losing the granularity and load balancing benefits of conditional division, the best work stealing strategies so far require a global knowledge of the queues occupancy [12] in order to select the best queue to steal from at any time. Proposed hardware support for work stealing similarly relies on a central hardware table for an efficient implementation of stealing [17].

In this paper, we show that it is possible to design a hardware support for conditional division that entirely relies on *local* control and information, i.e., no central resource or access to memory is required for probing or division. In a shared-memory multi-core using a network-on-chip as an interconnect (e.g., a 2D mesh topology), resource probing and task division can be efficiently implemented with simple modifications to the network interface and the router. The key difficulty is to decide when task divisions should be allowed without having a global knowledge of the multi-core occupancy. For that purpose, we *propagate* the amount of task division requests across the network in the same way *pressure* propagates through a gas. Numerous division requests occurring within a certain area of the network translate into a local *task pressure* increase. If neighbor nodes are free, these nodes will absorb the pressure by accepting the tasks and the local pressure will decrease, as can be seen in Figure 1(c,d). If they are not free, the pressure will gradually propagate through busy nodes to free neighbor nodes which can absorb it further away and the pressure will similarly decrease, after a longer delay. Only when some node is ready to accept a new task can the pressure decrease. A core cannot divide tasks (increase the pressure) as long as its local pressure is too high. In order to implement this strategy and considering that each core is connected to a single router, each router is augmented with a task queue whose occupancy concretely serves as the task pressure value. Tasks are passed from a task queue to a neighbor router along decreasing task occupancy (task pressure) gradients until a core can seize the task. In order to probe resources, a core checks the task occupancy of its local router, a low-latency request which statistically correlates to resource (cores) availability. We empirically show that this resource probing and division strategy provides an efficient and scalable implementation of conditional division, compatible with future many-cores.

In Section 2, we present the principles of our approach, the experimental methodology in Section 3, and the performance evaluation in Section 4.

2. DISTRIBUTED CONTROL FOR PROBING AND DIVISION

In this section, we present our overall probing and division scheme. Figure 2 provides an overview of its operation. The distributed control for probing and division is based on a division-aware network in which the routers are each augmented with a task queue. The latter contains divided tasks resulting from successful probings/divisions, and which are in the process of migrating to a host core to start execution. Fast probing is achieved by having the core only consult its local router's task queue, without any need for communications among cores. The task queue occupancy is used as an indication of the local multi-core workload and determines whether probing is successful (division granted) or not. At division time, a new task is not assigned to a host core but rather starts traveling the network, dynamically finding a router (and thus a core) which has a locally minimal pressure. Traveling tasks thus propagate and balance task queue occupancy (pressure) across cores over time.

2.1 Probing

In order to support fast probing, a one-bit link connects the router to its local core through the network interface and provides a binary `div_en` (division enabled) signal to the core, indicating that the task queue has free slots. The left part of Figure 2 sums up the interactions.

In the core, probing starts with the software `capsule_probe()` call. This call translates into a combination of a load and a branch instructions. The load reads an I/O register containing the `div_en` signal. A value of 1 indicates a task queue occupancy lower than a preset threshold (50% in our experiments), where division is allowed because the

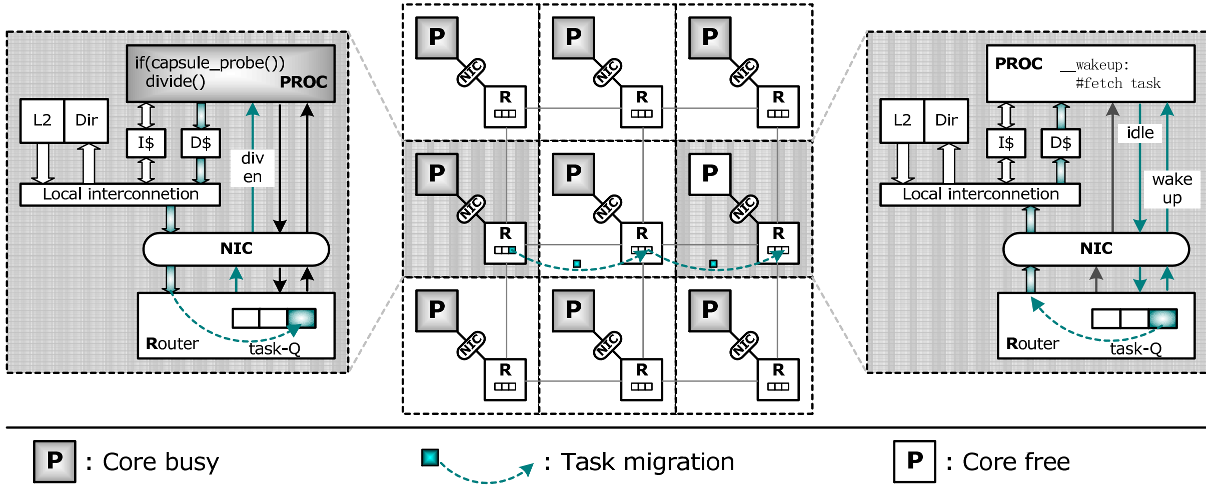

Figure 2: *Overview of the probing and division hardware support.*

local task activity is considered low enough. The branch instruction jumps to the division routine if the condition is 1. Since the load requires no memory access, it is always performed in a single cycle. Moreover, most probes fail (many more probes than divisions, usually) and the branch can be predicted as not taken with a high probability of success. Overall, both instructions require 2 cycles most of the time, which make them compatible with frequent probing, including within innermost loops.

2.2 Division

Task division is not synonymous with thread creation. Since thread creation can take 100K cycles or more (e.g., Posix thread creation on INTEL 2.4 GHz Xeon, 2 cpus/node) [7], it would be inefficient to create a thread upon each division. A division consists in passing all information required to set an execution context and restart a thread from a pool of statically allocated threads. Four data must be passed: a task id, a function pointer, a pointer to the function arguments, and a group pointer (used for synchronization, later explained in Section 2.5). Figure 3 shows the division code for QuickSort. When the probe succeeds, the division primitive `capsule_division` is called with the function pointer and the arguments pointer. The division primitive also collects the task id and the group pointer, and writes all the four data to the task queue. All these writing requests to the network interface are performed using standard store requests to memory-mapped I/Os. In order to ensure that the task is sent to the network without delay and consequently is executed by an available core as soon as possible, these store requests are not cached. The observed mean division overhead is 24 cycles.

In order to speed up the task transmission in the network, all task information is packaged into a single network packet in the network interface. Figure 4 (a) shows the task packet structure; it contains four flits, each flit is composed of 3 elements: a 1-bit packet type (task/data), a 2-bit flit type (head/body/tail), and a fixed size payload.

The task packet differs from the data packet, see 4 (b), in several ways. The task packet does not need the header flit to store the destination node because the tasks dynamically migrate across routers and the target node is not known a

```
// Probe for available hardware
if( capsule_probe() )
{
    // Perform division and quicksort on the right sub-array
    capsule_divide( (void *)(void *)qs_base_wrapper,
                   (void*)alloc_qs(right+1, inRight, arry) );
}
else
{
    // Division denied, carry on sequentially
    qs_base(right+1, inRight, arry);
}
// Quicksort on the left sub-array
qs_base(inLeft, right-1, arry);
```

(a) Quicksort parallel code

```
void capsule_divide (void *(*start_routine), void *args)
{
    uint32_t task_id;
    uint32_t *group_pointer;

    // get task id and father group structure pointer for sync
    capsys_divide(&task_id, &group_pointer);

    // write the task structure to I/O mapping NIC
    (*(uint32_t *) DIV_TKID_ADDR ) = task_id;
    (*(uint32_t *) DIV_GRID_ADDR ) = (uint32_t *)group_pointer;
    (*(uint32_t *) DIV_ROUT_ADDR ) = (uint32_t *)start_routine;
    (*(uint32_t *) DIV_ARGS_ADDR ) = (uint32_t *)args;
}
```

(b) Task division primitive

Figure 3: *Task division example code (QuickSort).*

priori. Moreover, since the task packets are only transferred between neighbor routers, they are only handled at the link level, and therefore, each flit must be tagged with its type (head/body/tail). On the other hand, data packets, which include both request and response data packets, do not need two bits per flit to distinguish between head/body/tail, because they are processed at the network level and all body flits follow the same path as the header, even if some other flits are interleaved. Finally, task packets and data packets use different flit sizes. Although this is not usually feasible, it is viable in this case because task packets are stored in the task queues, whose slots are appropriately sized. Additionally, network links are wide enough to transfer a flit from a data packet in a single clock cycle. This implies that some wires remain unused when transmitting task packet flits, but

this bandwidth waste is irrelevant because task packet traffic is scarce.

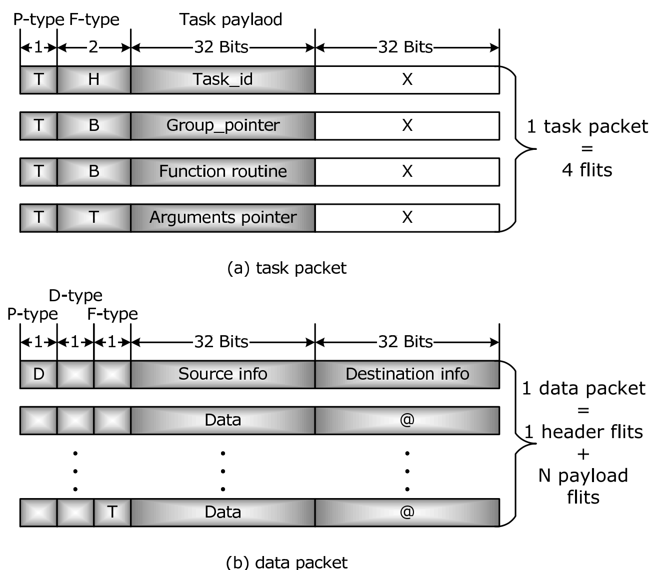

Figure 4: ***Two NoC packet structures.***

The task queue can simultaneously receive tasks from both the local core and the neighbor cores, but one task slot is always reserved for the local core. So, provided this slot is free at the time of probing, the task queue guarantees that it can receive the locally dividing task after signaling `div_en` without requiring a queue reservation mechanism.

2.3 Migration

The router task queue is a double-ended queue. The head tasks are the oldest tasks and the tail tasks are the most recent tasks. The local core retrieves the tail tasks from the queue, i.e., the most recent instead of the oldest tasks. This may seem counter-intuitive but we empirically found this strategy to be the best because tail tasks are often child tasks of recently run tasks and will often use data from their parents that most likely are still sitting in the local cache. Therefore, it turns out that it is a better trade-off to privilege cache locality over fast execution upon migration. Still, the head tasks migrate first, i.e., the older the task, the faster it migrates. A task that migrates will in turn be placed at the tail of the receiving task queue and potentially be executed more quickly by the connected free core.

Strategy. The task migration strategy aims at propagating tasks from regions of the network with high activity (numerous task divisions) to less active regions.

Ping-pongs. The naive migration strategy, illustrated on a 2D mesh, is the following: if a router R_{21} (R_{xy} with x indicating the mesh row and y the mesh column) contains N_{21} tasks and a router R_{22} contains N_{22} tasks with $N_{21} > N_{22}$, then a task from R_{21} can migrate to R_{22}. However, since the task queue size is small (we varied it from 1 to 5 slots in our experiments), it often happens that $N_{21} = N_{22} + 1$. After migration, $N_{22} = N_{21} + 1$, and within the next few cycles, the task can migrate back from R_{22} to R_{21}, inducing *ping-pongs* between the two routers. In order to avoid such ping-pongs, we introduce a safeguard where migrations are allowed only if the task queue occupancy difference is greater or equal to 2, i.e., $N_{21} - N_{22} \geq 2$.

Definition of pressure. While the task queue occupancy is a good measure of the local core activity, it has two weaknesses. First, it does not take into account whether the local core itself is idle or not. Typically, if the core of a router (e.g. R_{22}) is idle, it should take a new task immediately. If there are tasks waiting in its local task queue, it just executes one; if there is no task, the neighbor containing ready tasks (e.g. R_{21}) shall move a task to the idle core, whatever the task occupancy difference between them. As a result, we define the pressure as follows:

$$f(p) = 2 \times \texttt{core_busy} + \texttt{N_slots} \quad (1)$$

In the equation, `core_busy` is 1 when the core is busy and 0 when the core is idle, `N_slots` equals to the number of busy slots in the queue, and the factor 2 before `core_busy` aims at overriding the "difference of two" constraint if the local core is idle.

Second, a slightly more complex notion is to migrate the tasks along the most promising directions within the network, so that they "find" regions of lesser activity. In order to convey this pressure gradient, we want each router to account not only for its own local activity, but also for the activity of its neighbors. If that measure is included in the local pressure, when R_{22} sends its pressure to R_{21}, it would implicitly account for the pressure of its neighbors R_{12}, R_{23}, R_{32} (and R_{21} naturally). Finally, we define the pressure using the following equation 2:

$$f(p) = 2 \times \texttt{local_core_busy} + \texttt{N_slots} + \sum_{neighbors}(\texttt{core_busy}) \quad (2)$$

where the term $\sum_{neighbors}$ conveys the activity of the neighbor routers.

In order to propagate the pressure information across routers, each router is connected to all its neighbor routers with full-duplex status links, e.g., 2×4-bit links for a 2D mesh if the queue contains $2 \sim 9$ slots.

If multiple neighbor routers are eligible and have the same pressure, they are randomly chosen, using a fast low-power pseudo random generator [28, 4]. Conversely, a router only accepts a single task from other routers every cycle.

Tasks propagate through the network until they reach a region where few tasks are generated. Consider the example in Figure 5: in (a), router 22 contains 3 tasks and has the largest pressure; it chooses neighbor router 21, which has the lowest pressure, as the migration target; after task migration, a local load-balance is achieved in (b). Therefore, if there is a division hot spot occurring in one region of the mesh, all other regions will eventually see it through a rising pressure level. While the pressure level of any router is not necessarily instantaneously correlated with the global multi-core load, it will eventually be correlated after tasks have propagated. As a result, this local activity metric tends towards a global activity metric over time.

Possible limitation and solution. In the special case where only a single task is responsible for directly creating all tasks in the network, the presented migration strategy remains limited because it does not propagate tasks beyond a radius of $f(p)$ cores in each direction. However, there are two reasons why this limitation has little consequence in practice.

First, in a 2D network, with as little as 2 slots/queue, the propagation radius limit $f(p)$ is equal to 6, which covers $(1 + 4 \times \sum_{i=1}^{f(p)} i)$ cores, i.e., 85 cores, which is well beyond current and upcoming multi-cores. Second, in practice, it is unlikely that only a single task/core will create all tasks; in most cases, the child tasks create more tasks themselves, which spreads out task divisions. We did not empirically observe this situation/limitation for our target benchmarks and multi-core configurations.

Still, for future work, we plan to investigate alternative migration strategies with no such limitation. The source of the limitation is the "difference of two" constraint, which is itself motivated by ping-pongs. We plan to investigate alternative migration strategies with no such "difference of two" constraint, which would only attempt to reduce, but not eliminate, ping-pongs. To carry on with the physics analogy, the principle is to emulate "Brownian particles motion" where tasks randomly seek their way through the network, but are nonetheless statistically guided by lesser pressure gradients. For that purpose, we would randomly select the target router using a non-uniform random distribution based on the pressure of immediate neighbors. We could further reduce ping-pongs by monitoring the number of migrations of each task, and slowing down highly migrating tasks. However, as mentioned before, we saw no empirical justification for evaluating such more complex migration strategies for now.

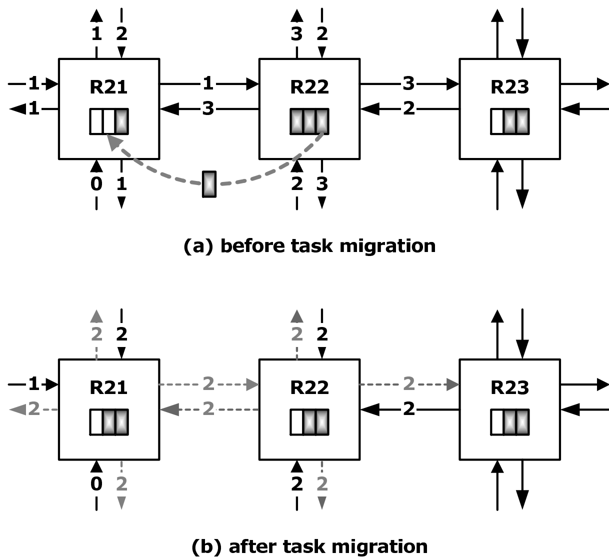

Figure 5: ***Task migration and load balance.***

Transfers. There is a cost/performance trade-off associated with tasks transfers. Tasks must be propagated quickly so that the overhead of parallelism remains minimal and the overall scheme is beneficial even to small tasks. At the same time, a dedicated routing network for tasks transfers would be overkill as the number of tasks transfers remains small compared to the amount of data movements.

We chose to use a special virtual channel [19] to transfer tasks to neighbor routers. Virtual channels are commonly used for Quality-of-Service purposes in NoC design [26]. They enable sharing a physical resource between logically independent channels. In our case, the shared resources are the inter-router wires/connections and the intra-router switches. In a conventional virtual channel implementation, multiple buffers are added for each input port in order to implement connection multiplexing among different types of transactions. In our case, there are two types of transactions: data and tasks. For the data transaction, two virtual channels are implemented (described below) in a conventional way, i.e., each input port has two data buffers. However, for the task transaction, only a single buffer is added (the task queue), shared by all input ports. This approach both simplifies the task arbiter logic and keeps the implementation cost low.

Figure 6 illustrates our division-aware router architecture. Each input port has a dedicated parser to detect whether the current packet contains a task and to distribute the input flit to the corresponding virtual buffers. The parser is just a simple 1:3 demultiplexer, the select bits are the 1-bit packet type and the 1-bit data type described in Figure 4. If the flit is found to have a task payload, it is sent to the task queue, otherwise it is further distinguished by the *D-type* bit and sent to either the request data buffer or the response data buffer. A 5:1 multiplexer is implemented before the task queue. The switch routing arbiter gives priority to the task queue over normal data buffers, ensuring fast propagation of tasks packets; the arbiter even interrupts data packet transmission when a task can be propagated.

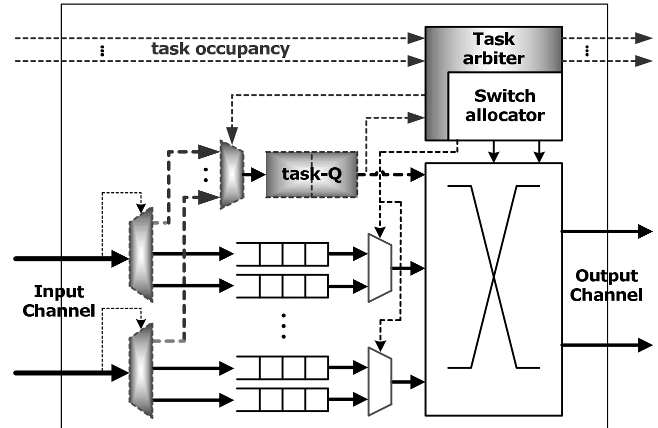

Figure 6: ***Division-Aware router architecture.***

The destination of a migrated task is determined dynamically, as the task progresses through the network, at each router/core and based on local task pressure. For that reason, we use a store and forward switching strategy [23] to transmit tasks packets, i.e., all the task packets must be in the task queue before initiating the next hop. The store-and-forward strategy can also avoid partial, and thus useless, task packets which would waste task queue slots. With store-and-forward, a packet can reside in at most 2 routers at any time. Each task slot in the queue can have any of the 5 states indicated in Table 1. The task migrates between the routers through the following protocol. When a router has `ready` tasks in the task queue and finds a target router for its head task, it sends the first flit of the task packet together with a request to the target router. If the request is accepted, the target router switches its task queue slot from `empty` to `receiving` and increases its task occu-

Task states	Description
empty	No task in the queue slot, router can receive a new task.
ready	The task in the queue slot can be executed or migrated.
receiving	The queue slot is receiving a new task.
executing	The task is being fetched by the local processor.
migrating	The task is migrating to the neighbor.

Table 1: ***Possible task slot states in task queue.***

pancy. The source router switches its task queue slot from `ready` to `migrating`. At the next cycle, the source router removes the first flit from the queue and sends one flit per cycle from then on. After the last flit has been transmitted, the source router switches its slot to `empty` and the target router switches the slot to `ready`. The task transmission latency from one router to a neighbor router is 4 cycles. If the request is denied, the source router will check again the neighbors occupancies and select another potential target router.

In the absence of tasks packets, the data packets naturally take advantage of the whole bandwidth. For data packets transmission on a 2D mesh, we use the conventional static x-y routing protocol and wormhole switching [1].

Deadlocks. In a multi-core system, even if the interconnection network implements deadlock-free routing, the cache coherence protocol can introduce a dependence between request packets (e.g., an invalidation request or a memory read) and response packets (e.g., an invalidation acknowledgment or a read response). Those dependencies can close the cycles and produce deadlocks. The simplest way to avoid this issue is by using a different physical or virtual network for request and response packets [18]. Thus, we implement two virtual channels per link, in addition to the special virtual channel mentioned above.

Because the task packets can follow any path (their path is not determined by the routing algorithm), they could potentially introduce deadlocks within task queues, which could become full with task packets cycling among them. In order to ascertain that this situation never occurs, we introduce the following restrictions:

1. A local task injection (upon task division) is limited by the `div_en` signal. A task injection is allowed only when the task queue has a free slot.

2. Task packet transmission is only allowed if there are 2 more empty slots at the receiving node than at the sending node, thus preventing queues from becoming full during the transmission.

3. Task packet transmission requests can never form a cycle because it would imply that every task queue in the cycle contains two more empty slots than the queue requesting transmission to it, which is impossible.

2.4 Execution

Upon receiving a migrated task, the local router of an idle core wakes it up. For that purpose, each local router receives a 1-bit *idle* signal from its core which indicates the core status. If the router local task queue is not empty and if the core is idle, the router picks the task at the tail of the queue, and sends the corresponding payload to the network interface. It then wakes up the core via an interrupt, which calls a system routine in charge of retrieving the task payload and initializing one of the statically allocated core threads, see Figure 2. The initialization routine sets the stack register to a pre-allocated stack to avoid dynamic stack allocation, then it retrieves the function arguments from the payload and initializes the corresponding registers, and sets the PC, see Figure 7. Traditionally, the compiler optimizes argument passing by using several registers instead of the stack when there are few arguments; additional arguments are passed by the stack. The thread will then start the function encapsulated in the payload. If this function is not in the local core instruction cache, the corresponding instructions will be fetched from shared-memory.

```
    .global _wake_up
_wake_up:
    lis     30,FETCH_ADDR    # get I/O mapping NIC address

    lwz     3,0(30)          # get TASK ID
    lwz     4,4(30)          # get GROUP POINTER
    lwz     23,8(30)         # get ROUTINE
    lwz     24,12(30)        # get ARGS POINTER

    bl      thread_execute   # syscall for thread management

    mtlr    23               # lr = start routine
    mr      3,24             # r3 = args
    blrl                     # call routine
```

Figure 7: ***PowerPC assembly code to retrieve the task payload and execute the user thread routine.***

2.5 Synchronization & termination

The recursive task division principle used in conditional parallelization creates an opportunity for more flexible synchronization. If the spawning and execution phases are followed by a synchronization phase, as in Cilk, just completed tasks will have to wait until their longest running child task finishes execution. In order to avoid cores sitting idle in this phase, those tasks' contexts have to be saved so that child tasks may spawn other sub-tasks to speed up their own execution and global progress along the critical path. We thus implement *recursive synchronization* as a "distributed" scheme across the task hierarchy: tasks can wait for their direct children, which in turn can spawn new tasks and wait for them. Waiting tasks have their context saved to make their core available again for computation.

However, if done improperly, this mechanism can lead to an excessive amount of context switches. This would typically be the case if we used a POSIX-like construct where threads can only wait for a single thread at once with the join primitive, thus serializing the wait and having the parent task wake and sleep again on the next children.

In order to circumvent that issue, the run-time system implements the notion of a *group* of tasks: a parent task and its child tasks belong to the same group. Figure 8 illustrates this hierarchical task group synchronization scheme using parallel QuickSort; 22 tasks are created across 7 groups, separated by gray dotted lines. A counter is associated to each group which indicates the number of living tasks within the group. Upon a task division, the group counter is incre-

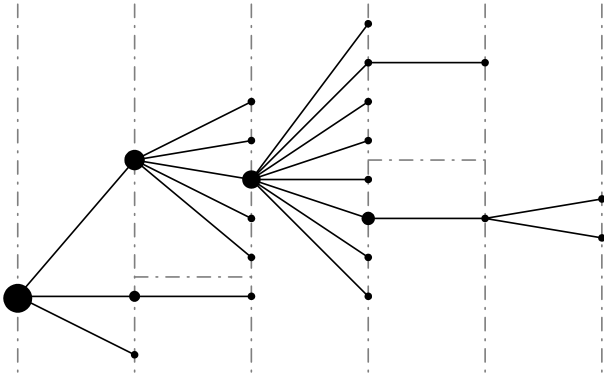

Figure 8: ***Synchronization groups and hierarchical synchronization: tasks are represented as black circles and divisions as edges.***

Parameter	Configuration
Core	PowerPC 405 32-bit RISC CPU with a scalar 5-stage pipeline
I-Cache L1	Private, 32KB, 4 ways, random replacement
D-Cache L1	Private, 32KB, 4 ways, random replacement, write-back
Cache L2	Shared and distributed across all tiles, aggregately 16MB,10-cycle latency for local core, $\sim$ 40 cycles for remote cores with 32 cores, on average
Memory	100 cycles
Cache Coherence	Full map directory protocol with write-invalidate inter-cache policy
Bus Latency	1 cycle
Task Queue	2 slots.

Table 2: *Tile architecture configuration.*

mented and a pointer to the group meta-data is passed to the child task. Conversely, when one of the child tasks completes its execution, it accesses the group counter through the group pointer to decrement it. When the seed task of the group finally reaches the join statement, the group counter is checked. If the seed task is the last living task in the group, then synchronization is complete. If there are still several living child tasks in the group, the run-time system saves the task context and recycles the core. Later on, as the last living task in the group finishes and decrements the group counter to 0, it wakes up the group seed task. There is thus at most one task context stored and 2 context switches per group (one to suspend the task and another to finally make it run again).

3. EXPERIMENTAL METHODOLOGY

Simulation and architecture. We use a cycle-level multi-core simulator based on the UNISIM [3] simulation framework. All benchmarks are fully simulated (no sampling/trace is used). Our architecture is a shared-memory multi-core, relying on directory coherence and the embedded 32-bit PowerPC405 core; the processor configuration is detailed in Table 2; we use 32-bit addresses in our simulator implementation. For experiments up to 32 cores, we implement a standard 4x8 mesh on-chip network. The network router architecture is described in Figure 6. Each router has 5 physical channels (ports) including the local processing unit channel. Each port of the router has two private virtual input data buffers both holding eight 67-bit flits. All 5 channels share one common virtual buffer for task transmission which can store two tasks, each of them holding four 35-bits flits. So each router has a total of $5 \times 2 \times 8 = 80$ data buffer entries and $2 \times 4 = 8$ task buffer entries. Each tile is an independent processing unit composed of a processor, an instruction L1 cache, a data L1 cache and a local interconnection (bus) and one bank of the shared L2 cache. The detailed parameters of the tile components are shown in Table 2.

The CAPSULE run-time system runs on top of the architecture, in each core, and all benchmarks rely on the run-time system primitives for probing, division and synchronization operations. The overhead of the run-time system activity is factored in all performance measurements. More precisely, we do not separate the run-time system and benchmark activities and consider the benchmark + run-time system as one workload and evaluate its overall performance.

Benchmarks suite. Adapting large existing parallel benchmarks suites, such as Princeton Parsec benchmarks [5], to a novel environment like CAPSULE or Cilk would require an excessive engineering effort. Moreover, the recent Berkeley roadmap [2] advocates *dwarfs* (implementations of algorithms) rather than full benchmarks as a more practical method for investigating novel parallel programming approaches. Since our parallelization approach especially targets applications with non-regular control flow and data structures, we follow the dwarfs approach and develop a set of relevant and well-known algorithms parallelized using CAPSULE. The algorithms are listed in Table 3. Their implementation ranges from 306 lines for `QuickSort` to 1665 lines for `Watershed`. For each program, we select several data sets (from 50 for QuickSort to 2 for Watershed).

In order to further analyze our strategy as a function of task granularity, we have developed a task generator where we can control the task granularity. We generate a constant number of tasks (10000) but we randomly vary their granularity. The task is a simple while loop (`while(i<n) i++;`), where `n` is randomly set. We distinguish between small-granularity tasks where `n` is varied between 1 and 10, and between 100 and 200 for large-granularity tasks. The division recursively splits the initial set of tasks. We control the random generation so that the total workload is constant.

4. PERFORMANCE EVALUATION

In Figures 9 and 10, we compare the performance of our hardware local division scheme against two software schemes: a central and a local software division scheme.

The central software scheme corresponds to the software shared-memory implementation of CAPSULE where all probing and division requests require access to a few shared variables stored in shared-memory. More precisely, probing is realized by reading a shared variable/counter corresponding to the number of available cores; if this counter is equal to 0, then the probing fails. In order to speed up probing, it is implemented as a two-step process: the shared counter is first accessed *speculatively*, i.e., without locks; if the counter is not null, then a second locked, and thus reliable, access is

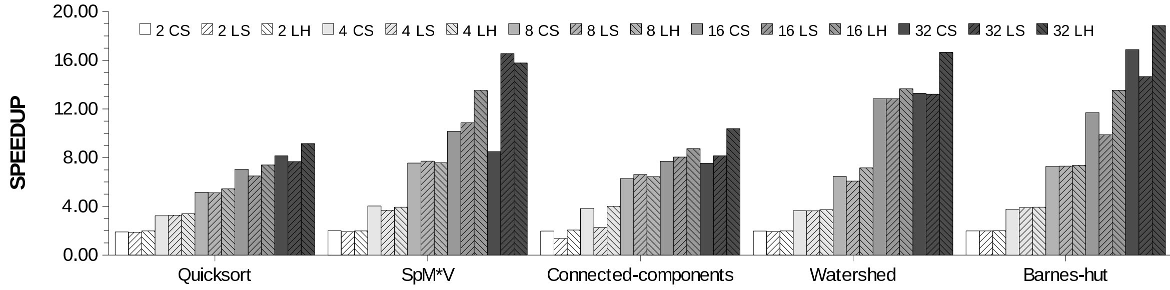

Figure 10: ***Central vs. local division schemes using benchmarks.***

Benchmark	Description	Lines	Datasets
QuickSort	The QuickSort sorting algorithm	306	50 arrays of 1,000∼500,000 elements
SpMxV	Sparse matrix-vector multiply kernel, the matrix is stored using the standard Harwell-Boeing format	342	40 sparse matrices from university of Florida collection
Connected-Components	An algorithm for uniquely labeling connected components of a graph by using depth-first search.	634	5 graphs with 10,000∼100,000 nodes
Barnes-Hut	A divide-and-conquer algorithm to find clusters of particles in the N-body problem.	815	2 datasets of 128 and 1,000 bodies
Watershed	An image segmentation algorithm that splits an image into areas and labels them, based on the topology of the image.	1665	2 images of 32*32 and 512*512 pixels

Table 3: *Benchmark suite description.*

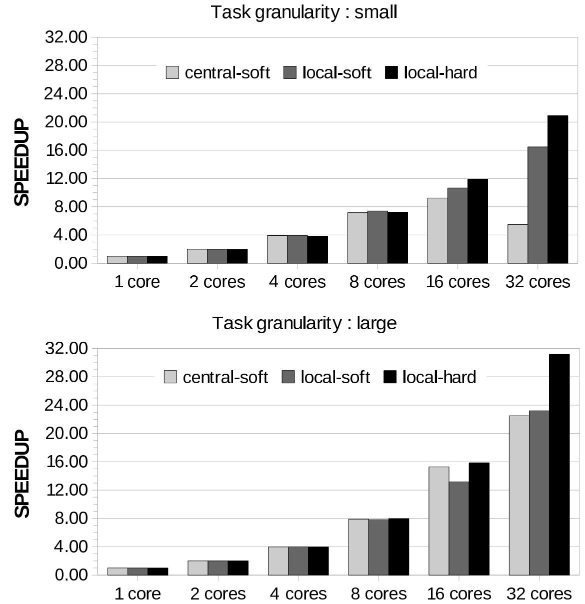

Figure 9: ***Central vs. local division schemes using the task generator.***

performed to validate the speculative information; the division then requires access to a number of shared variables for bookkeeping purposes. As the number of cores increases, the copy of the shared counter they contain is less likely to be accurate and it gets invalidated more often. Moreover, due to cache coherence, the copies of the shared counter need to be canceled or updated more frequently as the average division rate increases with the number of cores. Finally, the average latency to access the proper information through the network (shared variables required for probing and division) increases as well with the number of cores. As a consequence, while the central division scheme performs well for 4 and 8 cores [10], it does not scale as well as the local division scheme, as shown in Figures 9 and 10. Compared to the two local schemes, the difference is negligible for 2 and 8 cores, becomes noticeable for 16 cores and is significant for 32 cores.

The software local probe scheme is intermediate between the software central probe scheme and the hardware local probe scheme. A core only probes its neighbors, and the probing is done in software. For that purpose, each core maintains two variables. One variable corresponds to the core status (free/busy), while the other mirrors the status of the neighbor cores. The latter is updated by neighbor cores when it changes state, but since this update is not instantaneous, the variable cannot be considered to hold reliable information. When a core wants to probe its neighbors, it checks the status of the neighbor cores through its local variable in order to achieve a fast probe. If this first step is successful for one of the neighbor cores, it has to be confirmed by locking the neighbor core local status variable, confirming its availability and finally making the reservation. Only then can the division be initiated.

This scheme breaks the scalability limitations of the central software scheme, but it is significantly less efficient than hardware probing. Using the task generator, we find that the local hardware scheme is 299% faster than the central software scheme for 32 cores, and 32% faster than the local software scheme for small-granularity tasks (frequent probing), see Figure 9. The difference remains large for large-granularity tasks with respectively 38% and 20%. The results are further confirmed with the benchmarks, with a 35% average improvement over the central software scheme and a 20% performance improvement over the local software scheme.

We further highlight the scalability of the local probing/-division approach from 1 to 32 cores in Figure 11. A more

Bench	Parallelism	# divisions % divisions (on 1 core)	# divisions % divisions (on 2 cores)	# divisions % divisions (on 4 cores)	# divisions % divisions (on 8 cores)	# divisions % divisions (16 cores)	# divisions % divisions (on 32 cores)
QuickSort	18619	27 0.15	98 0.53	1031 5.54	2594 13.93	11976 64.32	18217 97.84
SpMxV	8204	55 0.67	134 1.63	520 6.34	560 6.83	1107 13.49	2122 25.87
Connected-Components	99751	3 0.00	8 0.01	48 0.05	65 0.07	126 0.13	235 0.24
Barnes-Hut	93	7 7.53	38 40.86	67 72.04	76 81.72	84 90.32	85 91.4
Watershed	243	5 2.06	11 2.23	43 17.7	62 25.51	89 36.63	195 80.25

Table 4: *Divisions (# of divisions and % of probings resulting in divisions).*

in-depth look at probings and divisions, see Table 4, helps better understand the remaining scalability limitations. For some programs, like `Connected-Components`, performance is limited by intrinsic parallelism (e.g., the actual number of connected components in a graph), hence the low division rate (0.24% for 32 cores). Note, however, that a low division rate is not necessarily synonymous with poor performance. Consider `SpMxV`, for instance: it performs well with only 25% divisions granted for 32 cores; unlike for `Connected-Components`, its division rate increases steadily with the number of cores, so that the low division rate only indicates significant unexploited potential parallelism rather than a lack of parallelism. At the same time, a high division rate is neither synonymous with high performance. `QuickSort`, with 97.84% division rate for 32 cores, scales well but in a limited manner. Since all sub-lists can be sorted independently, the `QuickSort` algorithm is intrinsically parallel; however, at each pivot step, the sub-list is again split into two parts, and this partition operation is sequential (scanning the whole list and comparing each element against the pivot). When the list is very unbalanced, this partition operation delays the creation of new sub-lists; artificially removing this delay improves performance by 156%, resulting in 23.5 speedup for 32 cores; so the performance limitations of `QuickSort` are tied to the algorithm.

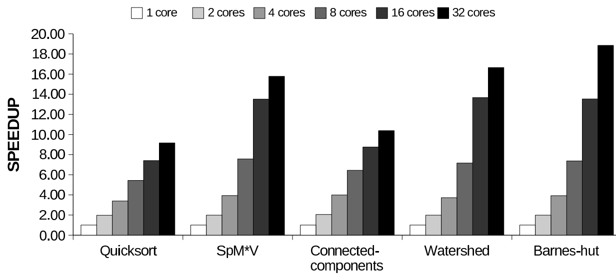

Figure 11: ***Scalability of local division scheme.***

We now investigate in more details the behavior of the local division scheme and its parametrization. In Figure 12, we indicate a distribution of the number of hops across the network of divided tasks for 32 cores. A significant fraction of divided tasks actually remain on the same core (0 hop, see `SpMxV` for instance): a task is divided, but because the parent task turns out to have very little remaining work, it quickly terminates after division, and the core fetches the tail task (of the task queue), which is often the recently divided child task, hence the 0 hop. As mentioned in Section 2.3, fetching the tail task is more efficient in such cases because the data for the child task already reside in the local cache; and, overall, we found that fetching the tail task is the better trade-off. As can be expected, tasks tend to migrate to neighbor nodes (high fraction of 1-hop migrations), but almost all programs do take advantage of long-distance migrations (2 to 5 hops); some programs like `Watershed` often send tasks farther away because they start with multiple active tasks, almost instantaneously creating high task pressure.

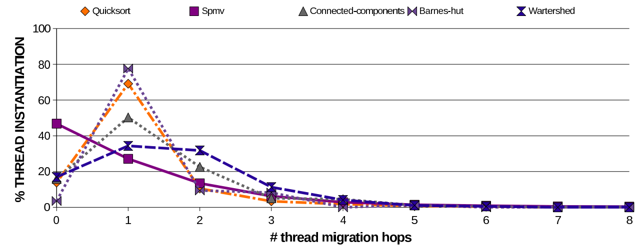

Figure 12: ***Task migration hops in the network (32 cores).***

In Figure 13, we investigate the impact of the number of slots of the task queue on performance. Note that for a high number of slots, the conditional division approach tends towards a work-stealing approach because a large number of divisions/Spawnings are allowed before resources are known to be available. Interestingly, we note that queues with a small number of slots tend to behave better for 3 out of 5 benchmarks, while the remaining 2 benchmarks are largely insensitive to the task queue size. When the queue size is large, tasks reside longer in queues and thus tend to migrate more easily and more often, inducing more long-latency data communications through the network. Small queue sizes have the advantage of imposing more stringent constraints on task divisions and thus task migration. At the same time, the overall scheme does take advantage of parallelism by dividing when necessary, and of the number of cores by allowing long-distance migrations when necessary, as mentioned before and in Figure 12. `QuickSort` and `Barnes-Hut` are rather insensitive to the queue size because of the low parallelization, see Table 4, and the division rate is high for these two benchmarks.

Finally, we investigate the impact of the migration strategy, particularly the decision to forbid migration to a neighbor core as long as the task queue occupancy difference with the target core is lower than 2. In Figure 14, we show that when an occupancy difference of 1 slot only is required, the performance is lower because of ping-pong effects, as mentioned in Section 2.3. After migration, the respective occu-

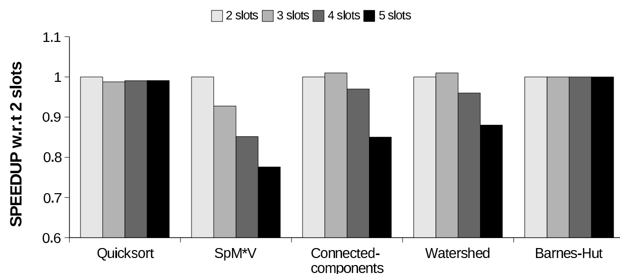

Figure 13: *Impact of task queue size.*

pancy of the source and target queues is reversed and the task could migrate back, instead of following the gradient of lowest occupancy across cores.

5. RELATED WORK

There has been substantial research on work dispatch strategies for regular multi-processors and cluster-like architectures, which led to the rise of the work sharing and work stealing paradigms [27].

Work stealing received much attention early on [15] [9] and became really popular with Cilk [8]. Cilk proposed a simple work stealing strategy that chooses the victim processor randomly across a whole machine and was proved efficient for a restricted class of parallel applications. In order to improve locality for SMP clusters, Distributed Cilk [22] biased steals so that they occur more frequently on neighbor processors. Later, a study of load-balancing on wide-area networks with the Satin environment [25] showed how to extend simple random stealing by performing local steals while a remote steal is pending, preventing the scheme performance from dropping as communication latencies increase. However, such mechanisms still eventually cause accesses to the task queues of remote processors, which increases network traffic and average communication latencies, especially with higher numbers of processors. Moreover, they may randomly put immoderate pressure on close nodes, leading to local traffic congestion.

DASH [24] also proposed hardware support for scheduling large amounts of tasks. It includes local task queues that pass tasks to a global queue when they overflow; scheduling is performed in a centralized manner, at regular intervals. More recently, NDP [11] introduced hardware support to reduce the communications and context switches overheads. Task and data queues reside in the routers to speed up data transfers. The scheduling policy, however, is still random stealing and has the same scalability limitations as mentioned in Section 1. Carbon [17] further improved upon the idea of managing tasks in hardware by grouping all the task queues into one hardware component, called the Global Task Unit (GTU), which enables fast work stealing. Moreover, it allows cores to prefetch tasks in order to reduce the GTU access latency. But it remains a central hardware component, with similar scalability limitations.

Our approach differs from these works in two ways. First, it relies on conditional division which removes the need for work stealing queues with unbounded length. The latter indeed requires a mechanism to store/fetch tasks into/from main memory when hardware queues are full. However, our scheme still provides good load balancing properties by taking advantage of resources as soon as they are available, while not clogging currently busy resources. Second, our approach waives any central hardware structure by strictly relying on local hardware support for managing task division and migration. The fast gradual propagation of tasks across the network is a form of global prefetching achieved through local control rules. Moreover, this scheme actually causes less task meta-data communications than a GTU, since tasks are routed to their destination directly instead of having to pass by the GTU first.

With respect to the NoC implementation, our hardware support shares several features with QoS hardware supports, used to guarantee low-delay jitter for real-time applications. AEthereal [13] defines a router-based NoC architecture that supports both guaranteed service (GS) and best-effort service (BES) by implementing two different routers. Each router has its own data buffer, with the GS router having a higher priority. The MANGO network [6] implements virtual channels (VCs) as separate physical buffers; GS connections are established by allocating a sequence of VCs through the network. Vellanki et al. [26] supports guaranteed throughput traffic by dividing the virtual channels between GS and BES levels. However all these works focus on communications, they do not consider core computations, task movements or load-balancing issues. RCA [14] uses a special low-bandwidth monitoring network to propagate congestion information among adjacent routers, in order to improve global network balance; still, it focuses on the communication aspects. In our architecture, the task queue behaves as a virtual channel from a communication standpoint, but it also assists the core and router for task division and migration decisions.

6. CONCLUSIONS AND FUTURE WORK

We present a hardware support for conditional division-based approaches to parallel programming. The hardware support essentially consists of low-cost add-ons in the routers of the on-chip network. This hardware support is shown to improve the performance scalability of the parallel division approach by entirely localizing the division decisions, and access to centrally stored information is no longer necessary. At the same time, the division approach combined with this hardware support is shown to outperform a central division scheme, even though the central scheme has a more accurate view of the availability of resources/cores within the multi-core architecture.

Future work will include coupling this division approach with a prefetching scheme for faster transfer of the data of migrating tasks. Moreover, the purely local decision scheme makes it compatible with distributed-memory architectures. We are also investigating high-level software representation of data structures which, coupled with our division approach, may let the run-time system and hardware support entirely drive the migration of data, providing the scalability benefits of distributed-memory architectures with almost the same ease of programming as shared-memory architectures.

7. REFERENCES

[1] K.M. Al-Tawil, M. Abd-El-Barr, and F. Ashraf. A survey and comparison of wormhole routing techniques in a mesh networks. *Network, IEEE*, 11(2):38–45, Mar/Apr 1997.

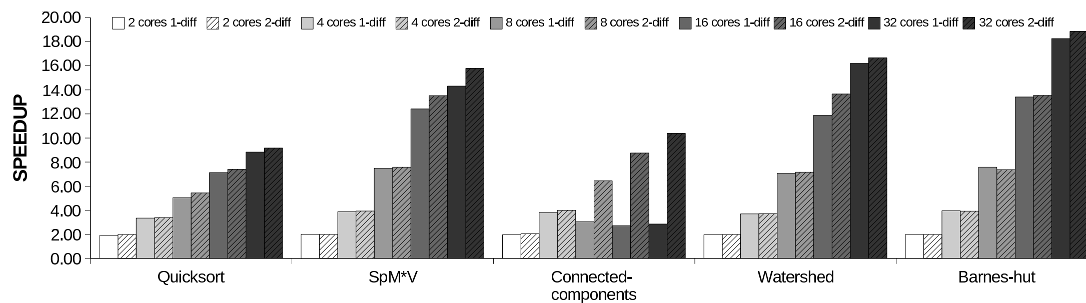

Figure 14: ***Impact of task migration strategy.***

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. Unisim: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Comput. Archit. Lett.*, 6(2):45–48, 2007.

[4] M.J. Bellido, A.J. Acosta, M. Valencia, A. Barriga, and J.L. Huertas. A simple binary random number generator: new approaches for cmos vlsi. In *Circuits and Systems, 1992., Proceedings of the 35th Midwest Symposium on*, pages 127–129 vol.1, Aug 1992.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[6] Tobias Bjerregaard and Jens Sparso. A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1226–1231, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Barney Blaise. Posix threads programming. https://computing.llnl.gov/tutorials/pthreads.

[8] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[9] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM.

[10] Olivier Certner, Zheng Li, Pierre Palatin, Olivier Temam, Frederic Arzel, and Nathalie Drach. A practical approach for reconciling high and predictable performance in non-regular parallel programs. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 740–745, New York, NY, USA, 2008. ACM.

[11] Julia Chen, Philo Juang, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh, and Margaret Martonosi. Hardware-modulated parallelism in chip multiprocessors. *SIGARCH Comput. Archit. News*, 33(4):54–63, 2005.

[12] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66, Sept. 2008.

[13] Kees Goossens, John Dielissen, and Andrei Radulescu. Aethereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, 2005.

[14] Paul Gratz, Boris Grot, and Stephen W. Keckler. Regional congestion awareness for load balance in networks-on-chip. In *HPCA*, pages 203–214, 2008.

[15] Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.

[16] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, 0:102, 2004.

[17] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, New York, NY, USA, 2007. ACM.

[18] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[19] Robert Mullins, Andrew West, and Simon Moore. Low-latency virtual-channel routers for on-chip

networks. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 188, Washington, DC, USA, 2004. IEEE Computer Society.

[20] Pierre Palatin, Yves Lhuillier, and Olivier Temam. Capsule: Hardware-assisted parallel execution of component-based programs. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 247–258, Dec. 2006.

[21] Chuck Pheatt. Intel®threading building blocks. *J. Comput. Small Coll.*, 23(4):298–298, 2008.

[22] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[23] Kumar Shashi, Jantsch Axel, Soininen Juha-Pekka, Forsell Martti, Millberg Mikael, ÃŰberg Johny, TiensyrjÃd' Kari, and Hemani Ahmed. A network on chip architecture and design methodology. In *ISVLSI '02: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Chulho Shin, Seong-Won Lee, and Jean-Luc Gaudiot. The need for adaptive dynamic thread scheduling. In *High performance scientific and engineering computing: hardware/software support*, pages 45–59, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

[25] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, New York, NY, USA, 2001. ACM.

[26] Praveen Vellanki, Nilanjan Banerjee, and Karam S. Chatha. Quality-of-service and error control techniques for mesh-based network-on-chip architectures. *Integr. VLSI J.*, 38(3):353–382, 2005.

[27] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):979–993, 1993.

[28] S. Zhou, W. Zhang, and N. Wu. An ultra-low power CMOS random number generator. *Solid State Electronics*, 52:233–238, February 2008.